PLEXC: A POLICY LANGUAGE FOR EXPOSURE CONTROL

by

Yann Le Gall

B.S.E. Princeton University, 2008

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This thesis was presented

by

Yann Le Gall

It was defended on

March 25th 2013

and approved by

Adam J. Lee, Department of Computer Science

Daniel Mossé, Department of Computer Science

Jack Lange, Department of Computer Science

Thesis Advisor: Adam J. Lee, Department of Computer Science

PLEXC: A POLICY LANGUAGE FOR EXPOSURE CONTROL

Yann Le Gall, M.S.

University of Pittsburgh, 2013

With the widespread use of online social networks and mobile devices, it is not uncommon for people to continuously broadcast contextual information such as their current location or activity. These technologies present both new opportunities for social engagement and new risks to privacy, and traditional static 'write once' disclosure policies are not well suited for controlling aggregate *exposure* risks in the current technological landscape.

Therefore, we present *PlexC*, a new policy language designed for exposure control. We take advantage of several recent user studies to identify a set of language requirements and features, providing the expressive power to accommodate information sharing in dynamic environments. In our evaluation we show that *PlexC* can concisely express common policy idioms drawn from survey responses, in addition to more complex information sharing scenarios.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

The popularity of online social networks has contributed to an unprecedented amount of personal information sharing. Moreover, the widespread use of mobile devices encourages the broadcast of *contextual* information from any location. For example, smart phone users can send their current location to social networks such as Facebook Places [1], Google+ [2], and Foursquare [3]. Furthermore, technologies such as CenceMe [19] can infer the current activity (e.g., running or dancing) from a smart phone's on-board sensors. Sharing location information through social networks has even led to burglaries and other crimes, as reported by Nick Bilton in an online *New York Times* article [6]. With so many ways to share personal contextual information, the task of protecting individual privacy is becoming more challenging. One important challenge is to maintain the utility of information sharing without sacrificing personal privacy. To achieve this equilibrium individuals need to do more than simply define a static disclosure policy once and for all. They must be able to specify flexible and adaptive policies that can manage the disclosure of personal information in the face of both typical and atypical access patterns. We refer to such policies as *exposure-aware*.

Motivation. Over the years, a large body of research literature has explored a variety of access control mechanisms and their policy language encodings. Existing policy languages have incorporated powerful features to group principals into functional roles (e.g. [27], [17], and [15]) delegate authorization decisions across security domains (e.g. [2], [4]) and even manage state changes during policy evaluation (e.g. [3], [21]). However, few sharing systems or policy languages have drawn upon large user studies to inform their design. As a

[1] www.facebook.com/places/
[2] plus.google.com
[3] foursquare.com

consequence, the resulting languages and systems offer a variety of interesting features, yet may not provide users with the functionality needed to address their real-world exposure concerns. By contrast, we carefully consider findings from several recent user studies within the exposure space [5, 24, 28] and leverage a variety of findings from these studies to provide insight into exposure perception and control.

For example, Schlegel et al. highlighted the importance of exposure feedback through an intuitive interface [28]. Additionally, Patil et al. discovered that certain factors, like the frequency with which location requests occur, are more important to users than other common factors, like the current time of the location request [24]. This is quite interesting, as few existing systems allow for controlling the frequency of requests, while several (e.g [18, 26, 29]) provide policy constructs for controlling disclosures based upon the day of week or time of day. Another important outcome of this study was the identification of several common concerns and policy idioms that are not typically associated with social engagement purposes, such as only sharing location during emergencies or with law enforcement personnel.

Our Contributions. Findings from these recent user studies reveal several ways to address shortcomings in current information sharing systems and their respective policy languages. To summarize a few, location sharing systems and their disclosure policy languages must be flexible enough to support users with diverse privacy concerns [5]. Furthermore, it is important to provide unobtrusive, ambient feedback about how users' data are being shared without necessarily revealing the identity of the requester [28]. Finally, policies should provide the ability to manage disclosure based on more than just common factors, such as the identity of the requester, but more complex policies may not be easily expressed [24]. To address these concerns we propose a new policy language *PlexC* whose functionality is based in large part on the needs voiced by the human subjects who participated in these studies. In doing so we make the following contributions:

1. We survey the recent research literature for human subjects' data regarding contextual information sharing and exposure control. Based on these findings we develop policy language and system requirements necessary for servicing the exposure control needs of users;

2. We develop a general system model for contextual information sharing systems that represents the features of existing logically centralized systems and is capable of modeling more user-centric systems that may appear;
3. We design a novel policy language, *PlexC*, that addresses the limitations identified in recent user studies and specify its syntax and semantics. We further discuss the query resolution procedure used by *PlexC*;
4. To evaluate the utility of *PlexC* we demonstrate that it is both capable of expressing a range of common policy idioms and can encode interesting real-world information sharing constraints specified by the subjects of several survey studies.
5. We design and build a prototype implementation of *PlexC* that integrates with relational databases and leverages the *tuProlog* Prolog interpreter.
6. We conduct a preliminary evaluation of the prototype implementation using a set of microbenchmarks designed to characterize the system's performance and scalability against increasing numbers of queries.

Outline. We start by defining the exposure problem and by identifying a set of language requirements that are motivated by recent user studies in §2. Next we discuss related work in §3. The syntax of *PlexC* is described in §4, and a prototype implementation is presented in §5. In §6 we evaluate the expressivity of our policy language against real user policies, interpret the findings, and discuss future work in §7. Finally, we conclude in §8.

2.0 BACKGROUND AND REQUIREMENTS

In this section, we first define the concept of *exposure* and introduce the relevant research challenges. Next we highlight the results of several recent user studies that explore aspects of the exposure-control problem space. We conclude this section by enumerating a set of exposure-control policy language features the need for which is highlighted by these studies and other works in the research literature.

2.1 THE EXPOSURE PROBLEM

Before describing our system model and how it addresses the exposure problem, we first explain what we mean by "exposure". Intuitively, a user's ideal policy for controlling access to their personal data is a moving target that is, at best, approximated by the policies and controls that the user puts in place to protect their information. To paraphrase an example by Schlegel et al. [28], a user may initially set a policy allowing her co-workers to access her location during normal work hours to facilitate in-person meetings. However, if she later discovers that her boss is accessing her location every 5 minutes to ensure that she remains in the office, she may become uncomfortable. This disconnect between the employee's model of appropriate sharing and the level of sharing allowed by the protections that she put in place leave her more *exposed* to external queries and analysis than she had anticipated.

The problem of exposure control is non-trivial, as exposure can be viewed as a function over a multi-dimensional space expressing a human sentiment. Some of these inputs may be unknown a priori as many contextual factors may influence a user's perception of exposure. For example, someone's notion of exposure may be influenced by the time of day, their

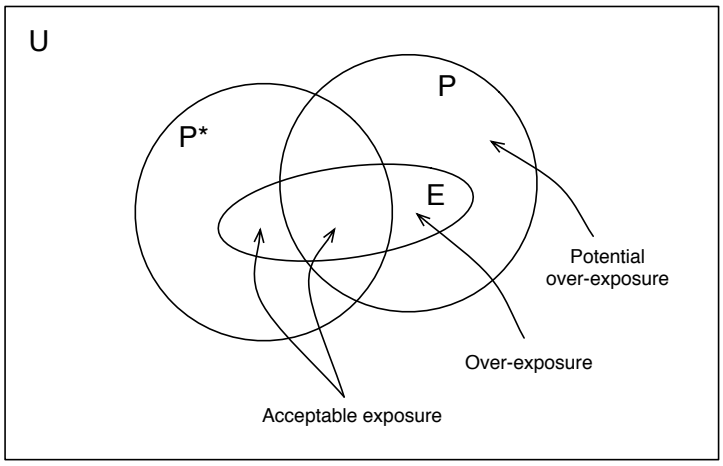


Figure 1: The exposure control problem.

current location, whom they are with, how many requests they have received, and so on. We propose that trying to control such a complex and dynamic property requires an adaptive process in which disclosure policies are continuously specified, enforced, and revised.

A semi-formal view of the exposure problem is captured in Figure 1. In this depiction U represents the universe of all access traces to a user's personal data. These traces describe sequences of queries to a user's data, which may be dependent on system and user context (location, activity, etc.), as well as other past queries. $P^\star$ represents the user's ideal model of data sharing—which is unlikely to be captured correctly due to the complexities of managing the myriad contextual facets of the exposure control problem—while P represents the access traces permitted by the user's deployed policies. E represents the access traces that have actually been made to the user's data and represents the user's *exposure*. $E \cap P^\star$ represents the user's *acceptable exposure*, whereas $E \cap P \setminus P^\star$ represents the user's *over-exposure*. $P \setminus P^\star \setminus E$ represents the user's *potential over-exposure*.

The goal of the exposure control loop is to ensure that E is confined to $P^\star$. In practice it may not be possible to achieve perfect exposure control, and thus the goal of the exposure control loop is to minimize the overlap between . As such, exposure can be represented as the set of traces $P \setminus P^\star$.

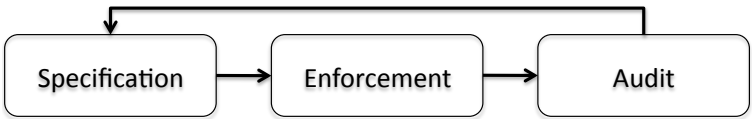


Figure 2: The exposure control loop.

The complexities of properly capturing $P^\star$ lead us to envision an *exposure control loop* in which a policy is specified and deployed, and feedback on the allowed access traces is periodically provided to the user. This process is depicted in Figure 2. This exposure feedback can then be used by the user to revise their policy over time, resulting in a sequence of policies allowing trace sets $P_1, P_2, \ldots, P_n$ that aim to minimize potential over exposure and avoid further over-exposure and ideally converge to $P^\star$. In the following sections we describe the design of the *PlexC* system and how it accommodates and encourages the exposure control loop.

2.2 RECENT STUDIES

Our work is grounded in a series of recent user studies relating to exposure control. Here we describe each study in more detail and explain how our findings are relevant to exposure control. The results of these studies contribute to many of the requirements discussed in Section 2.3.

When Privacy and Utility are in Harmony: Towards Better Design of Presence Technologies. In this study, Biehl et al. [5] explored user sentiment about presence data collection and sharing with an emphasis on workplace settings. Another goal was to explore the utility of receiving various types of presence information. They conducted a survey of 32 participants representing a wide range of ages, professions, and geographic regions across the US.

This study is relevant to exposure control because it measured how comfortable partici-

pants felt as a function of many variables, including the type of data collected (e.g., location, activity), the recipient of the data (e.g., boss, coworker, friend), the setting in which the data collection occurred (e.g., office, work event, home), and the format, owner, and location of the data collected. Also, the authors measured how comfort levels changed based on perceived utility for the recipient.

One important finding of this study was that comfort levels across different sensing technologies were bimodal. In other words there was no one-size-fits-all privacy policy that addressed all users' privacy needs. Thus, information sharing systems must be flexible enough to accommodate a wide range of user preferences. Another interesting finding was the strong correlation between how comfortable users were sharing information at a certain frequency/fidelity and their perceived utility of receiving information from others at that frequency/fidelity. Participants indicated that they felt more comfortable sharing their personal information at a rate that they imagined would be useful for the recipient. This suggests that negotiation and establishing mutual trust may be important factors in location sharing systems. Other important findings were that storing raw sensor data raises more concern that storing interpreted data or aggregated data. In general, there was a high concern associated with the location and ownership of stored personal information.

Eyeing Your Exposure: Quantifying and Controlling Information Sharing for Improved Privacy. Schlegel et al. [28] address the problem of location exposure feedback and control in a game-based simulated lab study. They develop and compare two different smart-phone interfaces: (i) a so-called "detailed information interface" that shows the number of requests in the last hour from different categories of people (e.g., friends, family, strangers), and (ii) a so-called "eyes interface" that shows the user a number of cartoon eyes on the main screen of the app in which each eye represents location requests from a single person, and the size of the eye grows depending on the number of requests and the social relationship.

This study is relevant because it quantifies the role of frequency in exposure control and embraces the notion that informative *feedback* is an important part of controlling exposure. However, feedback that has too much information might reveal the identity of the querier. Likewise, if feedback is too frequent or obtrusive, then it may annoy the user. The findings

of this study suggest that it is possible to benefit from feedback without sacrificing querier anonymity or usability.

My Privacy Policy: Exploring End-User Specification of Free-Form Location Access Rules. Patil et al. [24] used an online study to ask over a hundred participants to write location-sharing policy rules using everyday English. In addition participants were asked to rate and rank the importance of a number of factors that might influence location sharing, such as the identity of the person requesting the location, the current location, the frequency of requests, and the like. The research questions addressed in this work are also very pertinent to the domain of exposure control. The notion of exposure varies across individuals and across many other dimensions. This study measured the preferences of a large sample of individuals and allowed them to freely identify factors that contribute to over-exposure. Furthermore, the ratings and rankings give evidence about which location factors should be prioritized in location sharing systems.

There were a number of interesting findings. Participants indicated that the "time of day" and the "day of the week" of location requests were less important than the "frequency of receiving requests". This was unexpected because many modern location sharing applications provide time-based rules but do not provide frequency-based rules (e.g. [29], [1]). Furthermore, in general people had difficulty expressing coherent policies that controlled for all of the factors that they rated as being important. Finally, the authors identified several common themes in the free-form policies. Some of these include complete manual mediation of requests, temporary blocking, and sharing only in case of emergencies.

2.3 LANGUAGE REQUIREMENTS

Here we describe a series of language features that are important for the efficient and accurate expression of exposure control policies. These requirements are motivated by past work as well as the recent user studies previously described.

[1] www.google.com/latitude

Disclosure Negotiation. In open distributed systems it is impossible to specify the trust relationship between all pairs of individuals a priori. Negotiation allows strangers to build trust by exchanging credentials, information, etc. Negotiation has been identified as an important feature and used in several policy languages, such as RT [17], Cassandra [4], Binder [13], and ATN [16]. Negotiation is also important for understanding the reasons for which a request was made, and individuals are more likely to feel comfortable sharing their location if they believe it will be useful to the requester [5]. An example of this type of policy idiom is given below: *Share my city-level location with anonymous requesters, but if the requester is willing to reveal his identity then also share my street-level location.*

Polymorphism. Here we use polymorphism to describe a policy whose requirements change based on the user's degree of over-exposure. Schlegel et al. explore ways of estimating and representing this metric [28]. The following policy illustrates the utility of exposure polymorphism: *Share my exact location with family only when my current over-exposure level is low; Otherwise if my over-exposure is high, only share my city-level location.*

Side Effects. Side effects appear in policy rules and specify transactions that modify the authorization state of the system when the rule is satisfied. Olson et al. [21] argue that side effects are appropriate in large dynamic systems where maintaining ACLs is inefficient. Furthermore, role-based policy languages typically require updates to the authorization state as users activate roles. However, most modern authorization languages do not explicitly provide constructs to express state changes, so management of state changes must be hard-coded into system resource guards [3]. An example of a policy that would be more easily expressed with side-effects might be: *The first 3 location requests from an individual require my explicit approval, but subsequent requests do not.*

Aggregate Operations. Aggregate operators can provide users with summary information about the set of accesses to their personal information (the region labeled "E" in Figure 1) [20], which often includes operations such as SUM, COUNT, or MAX, over sets and multisets of tuples. Frequency-based policies rely on the ability to aggregate records in the audit log, and Patil et al. [24] showed that individuals believe that the frequency of requests is an important factor to consider when creating location sharing policies. Further-

more, it is demonstrated by Dell'Armi et al. [12] that aggregate operators can increase the modeling power of disjunctive logic programming languages and provide concise knowledge representation. A typical use of this feature would be the use of aggregation to limit the frequency of location disclosures, e.g.: *Do not share my location more than 10 times per day.*

Querier Privacy. "Querier privacy" often refers to anonymous access of resources, but, in general, it is not limited to protecting the identity of the requester. Querier privacy has been identified as an important feature in large online social networks (OSNs), especially those that have been used to organize protests and share sensitive documents [1]. Interestingly, Tsai et al. [29] showed that users of the location-sharing technology *Locyoution* felt more comfortable sharing their location when they were given feedback about who requested their location. The ability to provide users with exposure feedback might seem to be incompatible with querier anonymity, but Schlegel et al. [28] demonstrated intuitive feedback interfaces that accomplish this.

Delegation. Delegation allows disclosure decisions to be passed on to a trusted authority. The utility of this feature is apparent in the following policy rule: *Share my location with the same people with whom my friends share their location.* Delegation is an important feature of many authorization languages, some of which include RT [17], CTM [15], Secpal [2], Cassandra [4], Ponder [11], and Binder [13]. In large decentralized systems preexisting trust relationships often do not exist between authorizer and requester. Thus, delegation allows the authorizer to make decisions based on trusted third parties. Delegation also simplifies policies in hierarchical systems.

Groups and Roles. In role-based access control (RBAC), subjects are assigned to one or more roles (or groups), and permissions are assigned based on their roles, e.g.: *Family members can always see my exact location, but colleagues can view my location only during work hours.* RBAC greatly simplifies permissions management, is well suited for large organizations in the commercial and government sectors [27]. Recent studies have also shown that users consider groups and roles to be important factors when sharing their location [5, 24]. Furthermore, a recent study by Patil et al. reported that disclosing location with an unintended audience was one of the most popular causes of regret in users of location

sharing systems [25].

Time and Location-based Rules. Time-based rules control disclosure based on the current time. Similarly, location-based rules control disclosure based on the current location of the policy owner or the requester. As an example of a location-based rule, the owner might define a number of "named regions" as a coordinate pair and a radius, and associate regions with a sharing policy: *Share my location with family only if I am at the hospital.*

Time intervals and named regions are natural ways to specify policies that accommodate daily schedules and routines, and previous work has demonstrated that users of OSN's are comfortable expressing policies using these features [29]. Furthermore the current time and location of an individual influence the type of information that she is willing to share [5], e.g.: *Share my location only between 9am and 5pm.* As previously shown, policy rules that are based on the frequency of requests can also be implemented by combining features that allow access to the current time and audit log. Patil et al. observed that frequency-based rules may have a greater importance than time-based rules [24].

Disclosure Levels. In a policy language that supports multiple disclosure levels, the policy owner can specify the degree of information to disclose. For example, in response to a location request, the policy owner might choose to disclose only the name of the current city. This feature would accommodate many of the challenges identified by Biehl et al. [5]. They found people were more comfortable sharing detailed location information at work and less detailed information outside of work. Therefore, comfort with different disclosure levels is highly influenced by current location.

3.0 RELATED WORK

In previous sections, we looked at several user studies that shed light on some the desirable features and criteria for information sharing systems. We also talked about the challenges of meeting these criteria without endangering the privacy of users. In this section, we survey both well established policy schemes as well as several recent and feature rich policy schemes and illustrate that none of these schemes supports the full set of features outlined in Section 2.3 (Table 1 summarizes the features of these schemes). The body of literature describing access control policy languages and policy idioms is extensive and diverse: many languages draw their syntax or semantics from Datalog and are designed for distributed environments, [4, 13], while others are based on XML [30] or object-oriented paradigms [11]. Here we organize our review into the following categories: traditional access control models, logic-based and trust management approaches, and deployed location sharing systems.

3.1 TRADITIONAL ACCESS CONTROL

Researchers have developed role-based abstractions, (e.g. [17, 30]) to simplify the management of user rights. Damianou et al. introduced Ponder, an access control language for a variety of applications such as firewalls, operating systems, and databases [11]. In addition to traditional features such as roles and delegation, Ponder supports policies that require actions to be taken after being triggered by a certain event. Unlike many other authorization languages, Ponder is described as a declarative, strongly typed, object-oriented language.

Park and Sandhu introduced $UCON_{ABC}$, a family of models for usage control (UCON) [22, 23]. UCON is a conceptual framework that provides a comprehensive approach to managing

access control, Digital Rights Management (DRM), and trust management. It can express a wide variety of policies by applying different combinations of authorizations, obligations, and conditions to digital objects. For example, basic RBAC can be expressed using authorization rules alone, whereas DRM can be expressed using a combination of authorization rules, conditions, and obligations. UCON also explores the complexities that arise when data consumers become data producers, if, for example, a client's personal information is logged during transactions.

3.2 LOGIC AND TRUST MANAGEMENT

In addition to policy languages that have evolved from the access control approach, we examine languages inspired by trust management approaches, such as [7, 8], which combine the management of policies and trust relationships, as well as distributed logic-based approaches (e.g. [4, 13]), which can concisely and compactly manage very complex policies. While some of these approaches have a logical syntax and semantics, others are based on XML [30] or object-oriented paradigms [11].

Li et al. introduced the *RT* framework, which consists of a family of related languages for specifying distributed authorization policies [17]: *RT* is a role-based trust management language in which policies are constructed using four simple rule types that assign users to roles, represent delegations, and structure roles into hierarchical relationships; RT_1 extends this basic framework with support for parameterized roles; RT^T provides syntax for specifying policies that require thresholding and separation of duty; and RT^D introduces constructs for constrained delegation. *RT* has both a set-based semantics and a Datalog-based semantics, and policies can be efficiently evaluated via translation into a Datalog program.

DeTreville presents Binder, a security language for distributed systems, which is based on Datalog [13]. However, unlike basic Datalog, Binder programs can securely communicate with other Binder programs across distributed environments using signed certificates.

Becker et al. developed Cassandra, which is built upon Datalog with constraints (Datalog_C [4]). Cassandra provides role-based trust management in distributed domains with credential re-

trieval, separation of duty, and role activation/deactivation. Additionally, Cassandra rules may contain a constraint c drawn from a constraint domain C that can be tuned to provide different tradeoffs between computational complexity and expressivity. Becker et al. then build upon the extensibility of Cassandra in SecPal [2]. SecPal has a high-level natural syntax and its design features include delegation, constraints, and negation in queries. SecPal policies can also be compiled into Datalog$_C$ programs.

The State Modifying Policies framework [3] can be used to extend policy languages based on distributed logics with concepts from Transactional Datalog [9]. This provides support for the use of policies that are capable of adding/retracting facts to/from the policy's logic program at runtime. This is useful, e.g., for supporting policies that can augment and examine their own audit logs.

Recently, Gunter et al. described an idiom called "Experience-Based Access Control" (EBAM) [14]. Briefly, EBAM is a set of models, tools, and techniques to reconcile the differences between ideal policies and the operational policies enforced by the underlying system. A typical approach for realizing EBAM may include maintaining and analyzing an access log to suggest ways to update and improve existing rules. This iterative process is similar to the exposure feedback loop discussed previously; however, *PlexC* focuses on the human perception of exposure.

3.3 DEPLOYED LOCATION SHARING SYSTEMS

There are a number of location sharing systems that have been deployed as part of research projects, commercial ventures, and social networks. These systems are currently being used, yet they still lack all of the important features that we discussed in previous sections.

Facebook Places[1] is a location sharing system that has been integrated into Facebook, and it allows users to share their location with existing Facebook friends. It leverages the existing Facebook privacy controls, which is limited to access control based on groups and roles.

[1] www.facebook.com/about/location

FourSquare[2] is a location sharing system that allows users to check-in to at various locations. Users are encouraged to share their location with the chance to earn points, badges, and discounts at retailers. FourSquare provides a limited role-based access control model in which friends can view each other's recent location history.

Loccacino[3] is a location sharing system that originated from a research project at Carnegie Mellon University. Relative to other deployed systems, it provides a rich set of privacy features that determine if a location request will be permitted. Loccacino supports time-based rules and fine-grained role-based access control, allowing users to share their location with different groups, such as family members or coworkers[4] Loccacino also supports an innovative feature for location-based rules: users can draw a region on a map in which their location will be visible to others. Finally, users must request to see each other's location, and users can review a list of everyone who has made location requests.

3.4 DISCUSSION

In Table 1 we present a summary of features supported by different mainstream policy languages and sharing systems. Collectively, these policy languages introduce an impressive assortment of paradigms, idioms, and features for expressing security policies in different contexts. However, no single policy language provides support for all of the features identified in Section 2.3 as being important to the management of end-user exposure.

Note that popular social networks, such as Google+, Facebook, and Foursquare have very limited features. Their privacy settings are typically limited to one or more categories of friends (i.e. groups). It is possible that these systems intentionally decided to support a limited set of sharing features in hopes of offering a less complex user-experience. Furthermore, advertisers often provide incentives to generate and share content instead of encouraging privacy.

On the other hand, policy languages and systems from the research literature tend to

[2] www.foursqaure.com
[3] loccaccino.org
[4] http://loccaccino.org/laptop_instructions

Table 1: Comparison of language features.

	negotiation	exposure polymorphism	side effects	aggregation	tunable querier privacy	roles & delegation	time rules	location rules*	tunable disclosure granularity
Google+	no	no	no	no	no	yes	no	no	no
Facebook	no	no	no	no	no	yes	no	no	no
Foursquare	no	no	no	no	no	yes	no	no	no
RT	no	no	no	no	no	yes	no	no	yes
Cassandra	yes	no	no	yes	no	yes	yes	no	yes
SecPal	no	no	no	yes	no	yes	yes	no	yes
SMP	no	yes	yes	no	no	yes	no	no	no
Ponder	no	no	yes	no	no	yes	yes	no	yes
Binder	no	no	no	no	no	yes	no	no	yes
UCON	yes	no	yes	no	no	roles	yes	yes	no
Locaccino	no	no	no	no	no	yes	yes	yes	yes
PlexC	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**

*While other languages were not designed with location sharing in mind, they may be able to support location via minor extensions.

be more feature-rich. While some of the features that we list are not directly supported in other systems, it is possible that they could be included as language extensions or supporting libraries. This is not surprising, as exposure management was not a primary goal during the development of this prior work. However, post-hoc language extensions can risk complicating the performance and usability of systems relative to their original design.

Despite the fact that research based policy languages are more expressive than deployed systems, there are still many unsupported features. For example, few existing languages and systems include direct support for aggregation. This is a crucial ability without which it is not possible to limit the rate of sharing. Furthermore, for the systems that provide some form of feedback to the policy author, the identity of the querier is not protected.

In the next section, we describe *PlexC*, a policy language for exposure control that was designed not only to meet all of the needs identified in Section 2.3, but also to take into account the findings of recent user studies in the domain of exposure control.

4.0 PLEXC: SYSTEM AND SYNTAX

In the previous section we showed that existing policy languages and systems do not support an expressive set of privacy features that were identified as important for location sharing in recent user studies. We therefore develop the *PlexC* exposure control language, which is based on a subset of Prolog and includes a rich set of extensions. In Section 6, we provide further evidence demonstrating that PlexC is capable of meeting all of the requirements identified in the previous section. In this section, we now describe the *PlexC*'s system model including its components, interfaces, and assumptions.

4.1 MOTIVATION FOR DECLARATIVE LOGIC PROGRAMMING

Prolog is a logic based language with well-defined declarative semantics and efficient query evaluation algorithms. It provides a nice environment within which to express authorization policies. Indeed, many policy languages—including *PlexC*—are based on Datalog or can be translated into Datalog programs (e.g., [2, 3, 4, 13, 17]). *PlexC* uses an embedded Prolog interpreter to evaluate user privacy policies. However, *PlexC* only relies on a small subset of Prolog that more closely resembles Datalog, which is a logic programming language for deductive databases. We now provide a brief review of the terminology, syntax, and semantics of Datalog to familiarize the reader with the underlying programming model upon which *PlexC* is built.

Datalog is a syntactic subset of Prolog, and programs are composed of *facts* and *rules*. A *rule* is a statement of the form $q :- p_1, p_2, ..., p_n$, where q and each p_i for $1 \leq i \leq n$ are *literals*. Intuitively, this rule can be read as "p_1 and p_2 and ... and p_n imply q". q is referred

to as the *head* of the rule, and the *body* is composed of each p_i. A *fact* is a rule that contains only a head and no body.

A *literal* has the form $P(x_1, x_2, ..., x_m)$ where P is a predicate name followed by a tuple with arity m, and each x_i for $1 \leq i \leq m$ is a *variable* or *constant*.

Consider the following example that demonstrates a simple Datalog program:

```
friend(alice,bob).
friend(carol,alice).
connected(X,Y) :- friend(X,Y).
connected(X,Z) :- connected(X,Y),connected(Y,Z).
?-connected(X,bob).
```

This example demonstrates the simple inductive semantics of Datalog. New facts are derived from the head of a rule if existing facts can satisfy all predicates in the body of the rule. The first two statements are ground facts. These are sometimes stored in a physically separate database called the *Extensional Database* or EDB. The next two statements are rules, which are stored in the *Intensional Database* or IDB. The EDB and IDB contain disjoint sets of predicates; as such, predicates defined in the EDB may only appear in the body of rules, and may not appear in the head of any rule. The last statement above is a query that seeks to find all bindings of `X` such that `X` is an ancestor of `bob`. In the above program, the tuples `friend(alice,bob)` and `friend(carol,bob)` satisfy this query.

Pure Datalog does not allow negation, which can threaten the evaluation safety of a Datalog program. Typically, negation is handled using *stratification* or *closed world assumption* (CWA). Stratification imposes an evaluation order on rules where negated body predicates must be evaluated before predicates in the rule head. CWA allows the inference of negative ground facts if they do not appear in the EDB [10]. We use the CWA strategy for negation in *PlexC*.

4.2 PLEXC

We now describe the set of extensions distinguishing *PlexC* from Prolog. First, we discuss the interface across which external applications communicate with our system. We then explain how transactional updates to the system state can be expressed and how policy authors can both create rules that are based on changes in the system audit log and rules that are sensitive to user feedback. Finally, we list additional built-in predicates and functions.

External Interface. External applications, such as location- or presence-sharing applications, communicate with the *PlexC* system to determine if a certain resource of a user should be disclosed to the requester. This communication occurs through an *external interface* exposed to these types of applications that is composed of a set of predicates described below:

Built-in Predicates and Functors:.

- `location(User)` is a system defined functor that returns the current location of `User`. The location is represented as a coordinate pair and a radius.
- `location()` is a system defined functor that returns the current location of the current policy author.
- `canAccess(User, Requester)` `canAccess` is a system predicate that is invoked when a location request from `Requester` is received by `User`. Evaluation of this request inserts a record into the audit log.
- `remote(User1, User2, Term)` This predicate determines if User1 is permitted to query the specified term, Term, belonging to User2's KB.
- `accessCount(Duration)` This is a functor that provides users with a way to count the number of times that her location has been revealed within the specified duration. This functor takes a single argument, duration, which is a string specifying the time interval to consider. For example, `accessCount("2 days")` will return the number of successful access attempts in the past 2 days.
- `accessCount(Requester, Duration)` This functor is similar to the accessCount functor which takes only one argument. However, it also accepts another argument which is the

username of the requester. Instead of counting all access requests within the specified time interval, this functor only counts successful access attempts from the specified user.

- `insertState(Target, Term)` This predicate inserts the specified term, `Term`, associated with the specified target user, `Target`, into the user's local KB. This predicate always returns true.
- `getState(Target)` This functor retrieves the specified term, Term, associated with the specified target user, `Target`, from the local user's authorization state.
- `removeState(Target, Term)` This predicate removes the `Term` associated with the `Target` from the local user's authorization state. This predicate always returns true.
- `testState(Target, Term)` This predicate determines if the specified term, `Term`, is associated with the specified target user, `Target`, in the local user's authorization state. This predicate always returns true.
- `hour()` This functor returns the current hour of the day as an integer in the range 0 to 24.
- `hourBetween(t1, t2)` This predicate is true if the current hour is within the interval specified by t1 and t2.
- `day()` This functor returns the current day of the week as a string.
- `weekday()` This predicate is true if the current day is not Saturday or Sunday.
- `now()` This functor returns the current unix timestamp in milliseconds;
- `today()` This functor returns a term representing the current date.

By creating a set of policy rules, the policy author is free to define the conditions satisfying the predicates in the external interface.

When personal information is disclosed via the `canAccess` predicate, a transaction occurs in which a record of the access is inserted into the system audit log. The record contains information about the requester, the resource disclosed, the time of disclosure, and the level of detail (granularity) of the disclosure. Prior research has explored incorporating transactions into Datalog. Transaction Datalog (TD) is an extension to Datalog for executing transactions that modify the database as rules are evaluated. TD supports the classical ACID properties as well as other properties like transaction hierarchies, concurrency, and cooperation [9].

PlexC also supports the notion of *state effects* as introduced by Becker and Nanz [3]. Transactional Datalog composes effects using the sequential transaction operator "⊗" [9]. This feature allows users to express policies that require role activation, separation of duty, or other state-dependent operations. For example, the following rule allows requesters to access an individual's location only once:

Example 4.2.1

```
canAccess(X) :- not testState(X,seen), insertState(X,seen).
```

Note that instead of the "⊗" operator, *PlexC* simply places effects at the end of rules to achieve the same result. This works because any predicate that cannot be satisfied will cause evaluation of the rule to end before remaining predicates are evaluated.

Built-in Constants. *PlexC* also includes a number of constants that refer to resources that the user is not responsible for defining:

- `LOCATION` is a constant used to identify location resources;
- `CITY` is a constant used to specify the city-level of granularity for location resources;
- `ANYONE` is a placeholder that matches any user when scanning the audit log;
- `true` represents the positive truth value.
- `false` represents the negative truth value.

User Policies. Users can define facts and rules to control disclosure. As with basic Datalog this allows the easy creation of groups and roles. Example 4.2.2 demonstrates a set of basic facts and rules that a user might create.

Example 4.2.2

```
canAccess(bob).
canAccess(X) :- isMember(X, friend).
```

The first statement is a fact that explicitly gives Bob access to Alice's location information. In the next statement, Alice also allows her friends to view her location. Thus we see that with basic Datalog syntax we can easily implement a simple, static role-based access control model.

Remote predicates. Users can also specify *remote predicates* by providing an identifier as a prefix before the predicate name. With this feature we can encode policies that require delegation. In the following example, Alice delegates disclosure decisions to Bob:

Example 4.2.3

```
canAccess(X) :- remote(alice,bob,canAccess(X)).
```

Similarly, with these features we can express basic forms of disclosure negotiation and other rules that are *quid pro quo*. In the following example, Alice only allows another user to access her location if she can access his location:

Example 4.2.4

```
canAccess(X) :- remote(alice, X, canAccess(alice)).
```

Additionally, *PlexC* allows policy authors to constrain the information in the KB that is visible to other users. This is achieved with the built-in predicate, `canQuery(U,P)`, which allows another user U to query the predicate, P. For example, the registrar at a university might allow a teacher, T, to query for any student S enrolled in a course C that she teaches.

Example 4.2.5

```
canQuery(T, enrolled(S, C)) :- teaches(T, C), enrolled(S, C).
```

Handling Exposure. Users have the ability to write policies that depend on the current exposure conditions. As noted by Schlegel et al. [28], aggregation is an important prerequisite for achieving this behavior. Mumick et al. [20] investigate extending Datalog with aggregate operators. They show that Datalog can be efficiently extended with aggregate operators using magic sets and semi-naive evaluation algorithms, which provide good heuristics over the naive, bottom-up approach. In order to ensure the termination of Datalog programs, aggregate operators are subject to restrictions such as stratification [12]. In our case, we provide special built-in predicates that are restricted to aggregating over a logically separated set of facts and predicates. This restriction is sufficient for our purposes, as it allows *PlexC* policies to aggregate over the audit log, for example. The following demonstrates how aggregation over the audit log can be used to limit the frequency of location sharing to no more than 5 times per day:

Example 4.2.6

```
canAccess(X) :- accessCount(X,"1 day") <= 5.
```

Here, `accessCount(X,"1 day")` invokes a search of the audit log for the number of accesses by requester X within the past 24 hours. Other acceptable arguments for time interval parameter include "min", "minute(s)", "hr", "hour(s)", and "week(s)."

In addition to aggregation over the audit log, users can write rules that depend on their current exposure. Prior research suggests that several factors contribute highly to an individual's notion of exposure, such as the social relation of the requester [28], the frequency of requests [24], and the surroundings at the time of request [5]. *PlexC* provides the access to this information through built-in functions, predicates, and language features, making it easy to define custom exposure functions. For example, a user might define exposure levels to be HIGH if the number of requests by strangers in the audit log exceeds a certain threshold.

Example 4.2.7

```
canAccess(X) :- exposure("MEDIUM").
canAccess(X,CITY) :- exposure("HIGH").
```

Keeping the User in the Loop. In the study by Patil et al. [24], many participants expressed the desire to mediate all requests for their personal information. To this end we introduce a built-in function `prompt(X,R)` that prompts the current user to give requester X permission to access resource R. Other participants simply wanted to be notified for each request, so we define a similar function `notify(X,R)`, which notifies the user that requester X has accessed resource R.

Additional Features. There have been a number of extensions to pure Datalog, some of which *PlexC* incorporates. These include built-in predicates, functions, and negation [10]. *PlexC* includes support for basic equality, comparison, and arithmetic operators. These can be viewed as infix predicates except that the operands correspond to terms, and the result of the atom is evaluated by the underlying implementation and does not depend on facts in the local KB. The following rule demonstrates both a built-in function to test if the current day is a weekday, as well as the built-in greater-than operator, and stipulates that location requests are only permitted on weekdays between 9am–5pm:

Example 4.2.8

```
canAccess(X) :- weekday(), hourBetween(9,17).
```

PlexC also supports several predicates to create *named regions*, which are essentially locations on a map with an associated radius. The `region(NAME,LAT,LON,R)` predicate defines a region $NAME$ centered at the coordinate (LAT,LON) with radius R. The predicate `inRegion(L,NAME)` tests if the location L is within the region, $NAME$. Example 4.2.9 only allows members of a *student* group to access location when the user is on campus:

Example 4.2.9

```
region('campus', 40.2, -100.2, 1km);
canAccess(X) :- inRegion(location(), campus), member(X,student);
```

Furthermore, *PlexC* supports a limited form of negation. Pure Datalog does not allow negation because it can threaten the evaluation safety of programs; however, negation can be supported by adopting either *stratification* or the *closed world assumption* (CWA) [10]. Stratification imposes an evaluation order on rules where negated body predicates must be evaluated before predicates in the rule head. CWA allows the inference of negative ground facts if they do not appear in the EDB. *PlexC* uses the CWA to handle negation. Example 4.2.10 demonstrates a rule that uses negation to implement an exclusion policy:

Example 4.2.10

```
canAccess(X) :- not(member(X,enemies));
```

Finally, we support a set of built-in functional-symbols that may depend on the deployment environment. For example, a location-sharing application might contain a set of functional-symbols to perform distance calculations, e.g. `within(L1,L2,D)` would return true if $L1$ and $L2$ are within distance D. Similarly, functions that provide reverse geocoding would be useful, such as `cityOf(L)`, which would return the city of the location coordinate L.

4.3 RULE EVALUATION

In typical Datalog systems extensional facts are applied to rules in the intensional database to generate new facts until no new facts can be generated (a fixed point). This bottom-up approach is straightforward and can occur before handling queries. However, more expressive

languages do not take this approach to evaluate queries. One of the reasons is that certain special predicates and function symbols cannot be computed prior to receiving queries. For example, some predicates and constants depend on current time and location (e.g., `accessCount`, many types of rules defining the `canAccess` relation), while others require the user's interaction (e.g., `prompt`).

Therefore, instead of using bottom-up strategies, modern expressive policy languages employ memoized, top-down evaluation algorithms that combine the efficiency of goal-oriented approaches while avoiding the non-termination issues of standard SLD resolution used in Prolog [4]. This is the approach the *PlexC* takes. To illustrate this process, consider the following example:

Example 4.3.1

```
canAccess(X) :- weekday(),
   remote(alice, bob, friend(X)),
   accessCount(X, "1 day") <= 5.
```

Access to the current user's location is contingent upon several factors. First, the date is obtained and tested as a parameter of `weekday`. Next, the second literal is a remote predicate indicating that the requester needs to be a friend of Bob. A query is therefore sent to Bob's exported predicates interface, and if Bob allows the current user to query this predicate, and the requester belongs to the `friend` role, then a positive result is returned. The `accessCount` functor invokes a query on the audit log and returns the number of accesses by the requester (in the current day), and the last item tests that N is no more than 5.

5.0 IMPLEMENTATION

5.1 TOOLS AND TECHNOLOGY

In this section we describe our prototype implementation of *PlexC*. We begin with a overview of the tools and technologies that were used. *PlexC* relies on an embedded logic interpreter. We chose to use version 2.5.0 of *tuProlog*[1], which is a light-weight Prolog engine developed by researchers at the University of Bologna. We chose to use *tuProlog* for a number of reasons. First, it is written in Java and is advertised as being compatible with Android smart phones, and one of our future goals is to implement *PlexC* as a distributed policy evaluation system that is capable of at least partially running on mobile devices. Furthermore, *tuProlog* provides a well documented framework for adding language extensions and low-level interface into the evaluation engine. Finally, *tuProlog* is free and open-source software; *tuProlog* and related packages are licensed under the GNU Lesser General Public License agreement.

There are many free and open-source alternatives to *tuProlog*. GNU Prolog[2] and SWI Prolog[3] are well known Prolog engines. They are both written in C, and provides a native C interface. However, they are not advertised as being compatible with mobile devices. We found that GNU Prolog for Java did not provide the same quality of documentation as *tuProlog*. We also examined IRIS, a Java-based implementation of Datalog, but we found the top-down evaluation strategy of Prolog to be more appropriate for many of the language features that *PlexC* includes.

[1] http://tuprolog.alice.unibo.it/
[2] http://www.gprolog.org/
[3] http://www.swi-prolog.org/

5.2 SYSTEM DESIGN

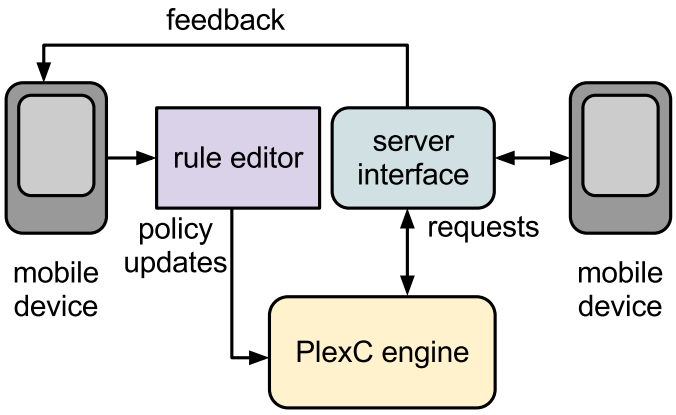


Figure 3: The integrated system model.

In this section we first describe the overall workflow and environment in which *PlexC* operates. This is shown in Figure 3. In a typical workflow, a *mobile device* makes a request to access to some resource, say, Alice's location. The request is handled by a central web server as an HTTP request, and is passes through the *server interface* to the PlexC *engine*, which runs as a service. The engine determines if the requester should be granted access to the resource. This decision might be based on information from several sources. In addition to evaluating disclosure policies, the evaluation engine may examine audit logs and the authorization state stored locally on the server. In a distributed architecture, scheduled for future development, the evaluation engine may even request information from remote *PlexC* systems. Finally, the disclosure decision is written to the system audit log, and the feedback component may decide to notify Alice about this interaction, and the response is relayed to the querier. After receiving feedback, Alice may wish to alter her policy using a user-friendly rule editor, which compiles a web form into the *PlexC* syntax and updates her policy, which is stored in the knowledge base.

The structure of the prototype implementation is illustrated in Figure 4. There are several components that we discuss here. The central component is called *PlexC Engine*. At a high level, it processes incoming requests and either makes changes to the system state or

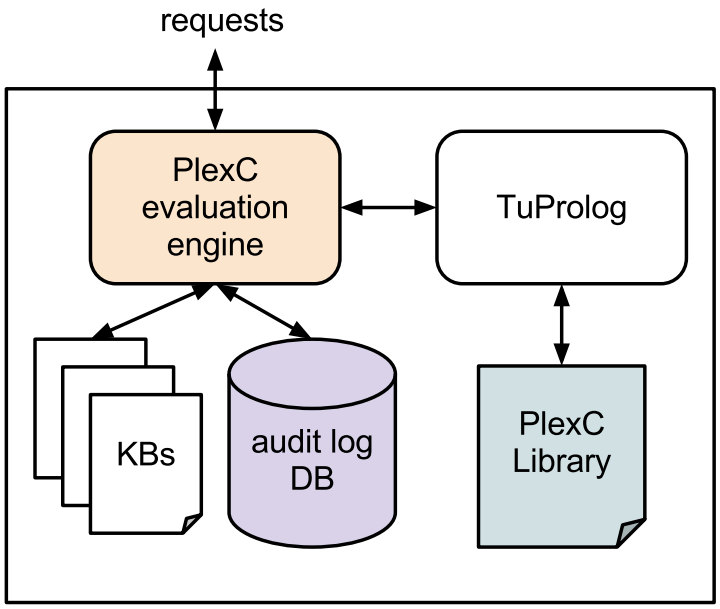


Figure 4: *PlexC* system components.

responds to the request with the appropriate information. Requests are issued for several purposes, including user registration, policy management, and requesting authorization. It is also responsible for managing the storage and retrieval of user policies. Policies for each user are stored as logically separate files on disk. The files are maintained as regular text files because *tuProlog* provides a convenient interface to load knowledge bases from regular text based representations. The engine loads a user's policy from disk when a request is received, given that the policy is not already cached in memory. The engine is also responsible for synchronizing user policies to disk when a user updates her policy.

PlexC makes use of a relational database to store certain data quickly and efficiently. These data include *audit log* records and transactional authorization state, and they are stored separately from the user policies to decouple data that are modified directly by the user from data that are generated programatically. Audit log records are highly structured and inserted into the a database table for every access request, regardless of whether the user policy permits it. Therefore, the user policy does not affect the retention of these records.

The audit log record contains the following attributes:

- `timestamp`: The time at which the request took place.
- `issuer`: The user initiating the request.
- `target`: The user receiving the request.
- `lat`: The latitude of the target user.
- `lng`: The longitude of the target user.
- `success`: A boolean flag indicating if the request was permitted.

PlexC also allows user policies to read and write authorization state that is not part of the user's intensional knowledge base. This state is written to a database table with support for transactions and does not clutter the user's privacy policy. This separation also ensures that policy evaluation completes even when negation is supported, as discussed in Section 4.1.

The `PlexC Library` is a set of functions that are loaded into the *tuProlog* interpreter at runtime. These functions implement the core features of *PlexC* that are not part of the Prolog specification. When the interpreter encounters one of these extensions in a user policy, it invokes the appropriate code in the PlexC Library. These features are described in the following section.

5.3 LANGUAGE FEATURES

Here we discuss how the language features of *PlexC* have been implemented. *PlexC* builds on top of *tuProlog* with a set of custom extensions to the core Prolog engine. This is accomplished by implementing a *library* class, which is loaded at runtime. The set of methods in the library provide the custom functionality. Methods with no return value are called *directives* in Prolog. Methods that return a boolean value are *predicates*, and all other methods correspond to *functors*.

Aggregation over the audit log. Aggregation over the audit log is an important feature for determining the exposure of a user. The audit log maintains a record of all access requests

for a given user, in addition to the time, location, and result of the request. One important metric by which exposure can be quantified is the frequency of access requests. The frequency of requests can be estimated by sampling the number of requests that have occurred within a certain time interval. *PlexC* provides a set of custom functors that scan the audit log and return the number of access requests that have occurred within a specified time frame. The function `accessCount/1(duration)` iterates over the audit log table, counting the number of access requests that have happened within the given duration until the present time. For example, `accessCount(‘‘2 days’’)` returns the total number of successful access requests that have occurred in the past 48 hours up to the present time. This allows a users to create policies that limit the rate of requests. The function `accessCount/2(issuer, duration)` is similar, except it only considers access requests that have originated from the specified issuer. This allows a user to create policies that limit the rate of requests from certain individuals.

Remote queries. PlexC provides a feature called remote queries, which allows rules from one user's policy to refer to the policy of another user. This behavior enables a number of important idioms, such as delegation of authority, quid-pro-quo policies, and trust negotiation. Remote queries are implemented with a user-defined predicate, `remote/3(issuer, target, term)`. Remote takes 3 arguments: the username of the current user, the username of the remote user, and the remote term to query.

For example, Alices policy might allow access to any friend of Bob:

```
canAccess(X) :- remote(alice, bob, friend(X)).
```

When Charlie requests access to Alice's information, the *PlexC* engine invokes the the remote predicate with the following arguments:

```
remote(alice,bob,friend(charlie))
```

The remote predicate first triggers a remote query to Bob's knowledge base to see if Bob allows Alice to issue remote queries on his knowledge base. This intermediate query has the following form:

```
canQuery(alice, friend(charlie).
```

If Bob's knowledge base does not contain this fact, then the remote query fails. Otherwise, the remote query continues, and if Bob's policy contains the fact `friend(charlie)`, then the remote query succeeds, and charlie is granted access by Alice.

State modification. In addition to the audit log, PlexC provides a table that user policies can read, write, and modify additional authorization state information. PlexC custom several predicates for this purpose. `insertState/2(target, data)` allows the policy owner to store arbitrary string data about a given specified user (target). `testState/2(target, data)` returns true if a pairing between the specified target and data exists. Finally `removeState/2(target, data)` allows the user to remove the state associated with the specified target.

Request Notifications. PlexC provides a predicate called `notify/1(requester)` which allows the policy owner to mandate the receipt of an email or text message notification upon successful evaluation of the rule. The single argument specifies a username to include in the notification message. For example, the rule `canAccess(alice, X) :- friend(X), notify(X).` will give X access to Alice's location if Alice says that X is a friend. Furthermore, it will send Alice a notification (e.g. email, text message) that X has successfully received access.

6.0 EVALUATION

In this section we give an informal evaluation of the expressiveness of *PlexC*. First, we show how the language design of *PlexC* gives it the expressivity needed to satisfy the language requirements outlined in Section 2.3. We then show how a variety of common policy idioms can be represented in *PlexC*, and we use *PlexC* to encode some of the more interesting policies gathered from free-response questions in the study conducted in [24]. Finally we conduct a preliminary performance evaluation by running microbenchmarks on a prototype implementation of *PlexC*.

6.1 COVERAGE OF LANGUAGE REQUIREMENTS

In Section 2.3 we enumerated several features of policy languages that are important for location sharing systems. Here we list examples that demonstrate how *PlexC* is expressive enough to support these requirements.

- **Groups and Roles:** Example 4.2.2 demonstrates how to define roles and limit access based on group membership. Policy authors can create different roles and assign membership relations using the natural Datalog syntax.
- **Delegation:** Example 4.2.3 shows how delegation is possible in *PlexC*. Delegation requires the evaluation of a relation whose records are not contained in the local KB.
- **Disclosure Negotiation:** Example 4.2.4 relies upon the evaluation of remote predicates to exchange information between the policy author and the requester until the conditions for disclosure are satisfied.

- **Side Effects:** Example 4.2.1 shows how *PlexC* draws upon existing syntax [3, 9] to specify changes to the authorization state during the evaluation of rules.
- **Disclosure Levels:** The amount of detail in a disclosure can be controlled by specifying the appropriate resource identifier. For instance, Example 4.2.7 shows how the granularity of information to be disclosed can be adjusted.
- **Time and Location rules:** Time-based rules can be expressed using the built-in constants that represent the current time and day as shown in Example 4.2.8 and Example 4.2.9, respectively.
- **Aggregate Operations:** *PlexC* provides support for aggregation over the system audit log via special predicates. This functionality can be seen in Example 4.2.6.
- **Polymorphism:** In Example 4.2.7 we encode a policy whose behavior is dependent on the target user's current level of over-exposure.

Some of these features are still under development and have not been included in the prototype implementation of *PlexC*. These include disclosure-levels, and user-defined functions for exposure polymorphism. We plan on developing these features in future implementations of the *PlexC* system, but we do not expect them to have a large impact on the performance characteristics of the system.

6.2 ENCODING FREE-FORM POLICIES

Here we further demonstrate the expressiveness and utility of *PlexC* by encoding some interesting and complex policies taken from participant free responses gathered during the study detailed by Patil et al. [24]. In this study, participants were first asked to rate and rank the importance of different location sharing factors, such as the identity of the recipient, the time of the day, and the current location. Next, participants were asked to write 5 to 10 location sharing policies using natural English language. We gave participants an example policies from the medical domain to avoid priming them.

The general quality of these *free-form* policies was lower than expected. Many policies were vague and consisted of incomplete sentences or single words, such as "FRIENDS"

and "BOSS." Other policies were more detailed but difficult or impossible to enforce. For instance, some participants wanted to prohibit disclosing their location to people with bad intentions.

Nonetheless, we were able to extract a large amount of useful information from the responses. Three independent coders reviewed the responses and classified the types of factors that were present. For example, the single word responses mentioned above imply that the identity or role of the recipient is important when sharing location. In addition to identifying important location sharing factors, we discovered additional policy idioms that were unexpectedly present in many responses. For example, a number of participants expressed the desire for complete mediation of all requests for their location. For example, one such policy was: *Keep my location private and ask every time someone wants to know my location.* This policy would have the following implementation in *PlexC*:

Example 6.2.1

```
canAccess(X) :- prompt(X).
```

Another common response was that users wanted to be notified after every location disclosure, but did not necessarily need to know who had accessed their location. For example, one user stated *"Any time anyone views my location, I must get a notification."* This rule is similar to the previous example, but does not require direct interaction from the user. One interpretation of this policy is that access should be allowed to everyone as long as there is a notification:

Example 6.2.2

```
canAccess(X) :- notify(X).
```

Another interpretation might be to modify any existing rules by adding a notification upon success, in which case the `notify()` predicate can be appended to the existing rules:

Example 6.2.3

```
canAccess(X) :- coworker(X), notify(X).
```

Location-based rules were also among the more unique responses. For example, one participant wrote, *"Allow users to see when I am within a particular radius of them."* An interesting implication of this is that the user wants to share her location only when it could possibly be of use to the recipient. This would be implemented in the following way:

Example 6.2.4

```
canAccess(X) :- within(location(X), location, "1km").
```

It is interesting to note that, by contrast, the complement of the above rule—i.e., allow access only when *outside* of a given radius—could be used to avoid stalkers or other unwanted attention.

Many participants also indicated that they would not want to share their location for social engagement purposes. However, many of these otherwise unwilling users of location-based services indicated that they *would* share their location during emergency situations, e.g.: *I would only want someone to know my physical location in an emergency situation.* This introduces the difficult problem of determining when the user is experiencing an emergency. However, one response provided some intuition: *"only if I'm missing for 24+ hours".* This policy could be approximated in the following way:

Example 6.2.5

```
canAccess(X) :- member(X, "emergency personnel"), accessCount("1 day") = 0.
```

By this encoding access is granted only if the requester belongs to the "emergency" role, and the audit log shows that nobody has received the user's location in the past 24 hours. The members of the emergency role could be defined by the policy author, or that duty could be delegated to some trusted agency.

6.3 PRELIMINARY PERFORMANCE EVALUATION

Real-world Policies:. Now that we have demonstrated that *PlexC* can encode policies obtained from user studies, we examine the cost of evaluating these policies. It is important to test the performance of evaluating policies that are typical for the average user. As indicated by the user study responses, users will often have a set of small, diverse rules for different social relations (e.g. family, friend, coworker) or different environments (e.g. work, home, gym). To craft so called typical policies, we incorporate elements from the free form responses of our user studies. For example, policies from the user study by Patil et

al. predominantly exhibited simple role-based rules for family and friends, in addition to role and time-based rules for coworkers [24].

In Policy 1, we present a complex policy in which the author, Alice, has specified 3 rules. The first rule states that any family member may access have access to her location. In addition to family members, Alice also allows any of Bob's friends to see her location no more than 5 times per day. This is captured in the second rule. Finally, in the third rule, Alice allows any employee to view her location no more than 4 times per day and only during weekdays between 9am to 5pm. Alice trusts her boss, Diana, to determine who is an employee.

Policy 1: Alice's Policy

```
family(sister).
canAccess(X) :- family(X).
canAccess(X) :- remote(alice,bob,friend(X)),
                accessCount(X,"1 day") <= 5.
canAccess(X) :- remote(alice,diana,employee(X)),
                weekday,
                hourBetween(9,17),
                accessCount(X, "1 day") < 5.
```

Policy 2 shows Bob's complete policy. It states that Charlie is a friend and that Alice may query for this fact.

Policy 2: Bob's Policy

```
friend(charlie).
canQuery(alice,friend(charlie)).
```

Finally, Policy 3 shows the complete policy of Alice's boss, Diana. It states that Eve and Alice are employees and that Alice may query for Eve's membership to the employee role.

The end result is that access to family members such as "sister" is restricted by a simple role based policy, while access to friends is restricted by role membership as well as the aggregate number of requests in the past day. Access to employees is governed by the most

Policy 3: Diana's Policy

```
employee(alice).

employee(eve).

canQuery(alice,employee(eve)).
```

complex rule, which checks for role membership, the number of aggregate requests, as well as the current time and day of the week. Therefore, this policy showcases some typical real world policies that vary in complexity.

In Table 6.3 we present the average time to query this policy for each of the different cases described above. The results show that the average query time is close to a millisecond.

sister	0.635 ms
charlie	1.630 ms
eve	2.087 ms

Table 2: real-world-test: Benchmark of real-world policy example.

In this section we describe the design of several microbenchmarks that demonstrate the performance of *PlexC* as the user policies grow more complex along different dimensions. Each of the microbenchmarks described below were executed on programatically generated policies, and they showcase the time required to evaluate queries against these policies. In each benchmark, we record summary statistics, including the mean, median, minimum, maximum, standard deviation. Care is taken to exclude the initial run from each benchmark due to warm-up costs. Benchmarks were performed on a commodity workstation running 32 bit GNU/Linux kernel version 2.6.38-13, with a 3.06GHz dual core processor with 4 gigabytes of physical memory.

Local Chains:. First, we evaluate the performance of *PlexC* with respect to the number of rules they contain. We refer to these policies as "local" because they do not contain reference predicates defined by other users, and therefore evaluation of a policy can take place locally within a single KB. Furthermore, these policies do not branch; their rules consist of chains of delegation. We measure the time required to respond to a request after it has been

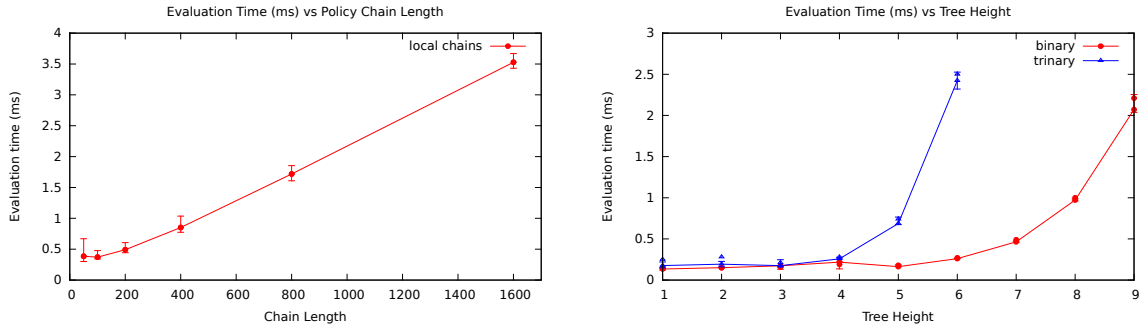
submitted. We expect a linear growth behavior, indicating that the the system scales well with a growing policy size. This test was also useful because it provided a baseline against which to compare remote query evaluation.

Figure 5(a) shows the results of the Local Chains benchmark. The number of rules in the policy varies along the horizontal axis, and the vertical axis represents the evaluation time in milliseconds. The figure shows the median of 1000 trials, and the error bars indicate the location of the first and third quartiles. We observe that the time required to evaluate a query grows linearly as the number of rules in the policy increase. In general, the query times are limited to a few milliseconds, even with policies that have over a thousand rules, which would not be representative for a typical user. This behavior is in line with our expectations. The policies evaluated in this benchmark do not branch and they do not refer to rules in remote KBs, therefore, there are no other sources of latency aside from the number of levels of indirection that the *tuProlog* engine must evaluate.

Local Trees:. Next we examined the scalability of query evaluation with respect to local policies whose rules contain more than one term in the body. Therefore, unlike the local chains, these policies have a branching structure. We expect that the time required to evaluate these policies grows exponentially with respect to the depth of policies, but we do not expect to see these types of large branching rules in the access control policies of the average user, as demonstrated by previous user studies [5, 24, 28].

Figure 5(b) shows the results of the Local Trees benchmark. Data was collected over 1000 runs. We tested policies with a branching factor of 2 and a maximum depth of 9, in addition to policies with a branching factor of 3 and a maximum depth of 6. Note that a binary tree of depth 9 has 256 leaves, and a ternary tree of depth 6 has 243 leaves. Both are unrealistic for a typical user policy. The data points show the median value and the error bars indicate the first and third quartiles. The figure demonstrates that the time required to evaluate a query grows exponentially with the height of the tree. The query times of the ternary tree grow more rapidly than those of the binary tree, as expected.

Figure 6 shows the evaluation time of queries with respect to the number of terms in the policy. The line representing the binary branching policy is close to the line representing the ternary branching policy. This is not surprising because the evaluation time primarily



(a) (b)

Figure 5: a) Evaluation time for local chaining policies. b) Evaluation time for local branching policies.

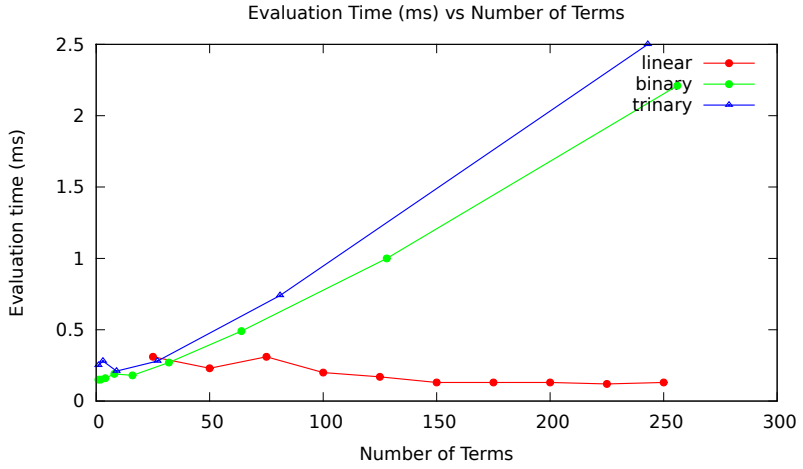


Figure 6: Evaluation time as a function of the number of terms.

depends on the number of terms that must be evaluated. However, the binary policy shows a slower growth rate than the ternary policy, and the evaluation times for the linear chain policy exhibit the slowest growth. This suggests that there may be a small performance penalty for policies with large branching factors. A possible explanation for this observation is that rules with more terms require more processing to combine the results of evaluating each term.

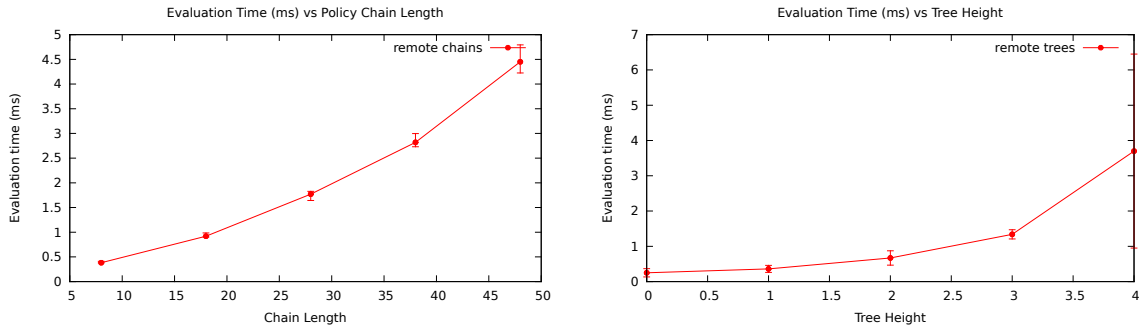
Remote Chains:. Here we examine the scalability of evaluation with respect to the number of remote queries in a policy. It is important to know the cost of remote queries because they are likely to incur additional cost as a result of data transmission across networks, organizations, or other physical or logical boundaries. In our benchmarks, these queries were sent and received on the same machine, but they had to travel across the external system API. To measure this, we record the time required to evaluate queries against policies that contain remote predicates (which are references to another user's policy). The expected result is that remote queries should grow linearly, but it should be more expensive than local chains.

Figure 7(a) shows the results of the Remote Chains benchmark, which was run for 1000 trials. The independent variable, on the horizontal axis, represents the length of policies chains. The dependent variable, on the vertical axis, represents the evaluation time. Policies lengths varied from 10 to 50. Again, the data points represent the median and the error bars show the first and third quartiles.

Remote Trees. Like remote chains, we also examine the time complexity of policies that both branch and make references to remote policies. The time requirements are expected to grow exponentially with the policy size.

Figure 7(b) shows the results of the Remote Trees benchmark, which was run for 1000 trials. The policies in this test had a branching factor of 2, and their depth ranged from 1 to 4. The independent variable, on the horizontal axis, represents the depth of the tree. The dependent variable, on the vertical axis, represents the evaluation time.

Aggregate Policies. It is important to understand the cost of policies that require aggregation over a growing audit log. The audit log will grow over time as requests are



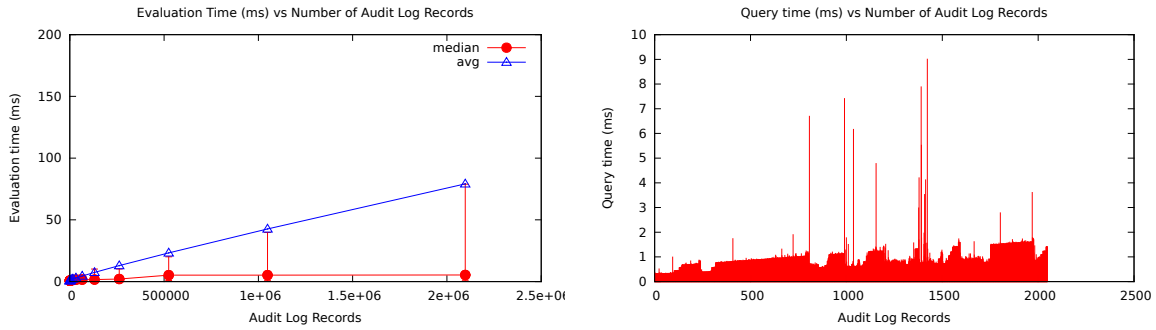
(a) (b)

Figure 7: a) Evaluation time for remote chaining policies. b) Evaluation time for remote branching policies.

received handled. In this microbenchmark, we measure the execution time of requests against policies that are based on frequency limiting rules (rules that require iterating over the audit log).

Figure 8(a) shows the results of the aggregate benchmark. This test was designed to measure the time required to evaluate the scalability of the audit log, which grows as access requests are processed. In this benchmark, the audit log is initially empty, and it is filled as mock requests are issued. When the audit log size reaches a power of 2, 1000 time measurement samples are taken. In this benchmark a query requires an entire scan of the all records in the database. The process continues until the audit log reaches a maximum size of 2^{20} records. The figure shows the average query time along side the median query time. The average query time shows a slow linear growth curve, as expected. Yet, the median query time shows a less pronounced growth rate, indicating that most of the queries are fast.

In Figure 8(b) we show the results of a performance profile of the audit log. We recorded the query times of over 2 thousand consecutive aggregate queries. Each bar in the figure represents the time for a single query. For the given table size, query times require less than 2 milliseconds. Like Figure 8(a), we notice a small increase in query times as the database grows. However, this test also reveals several outliers that take between 4 to 8 milliseconds.



(a) (b)

Figure 8: a) Evaluation time for aggregate queries. b) Audit log query time trace.

After these spikes, the query times temporarily decrease. One explanation for these outliers would be the various garbage collection, and housekeeping operations underway in the Java Virtual Machine and the database management system.

7.0 DISCUSSION AND FUTURE WORK

7.1 BENCHMARK RESULTS

Results of the microbenchmarks show that the time required to evaluate queries grows in proportion to the number of terms that must be evaluated in policy rules. So, for non-branching policies, the evaluation time is linear with respect to the number of rules. We also measured the cost of evaluating queries against policies with more complex branching rules, which contain more than one term to be satisfied in the body. In these tests, the evaluation time grows exponentially with respect to the number of levels in the policy-tree. This behavior is in line with our expectations.

These microbenchmarks are promising because they test the performance of *PlexC* against policies that are expected to be significantly more complex than the policy of a typical user. Even with policies that have over one thousand rules, we observe that query times are limited to a few milliseconds. Furthermore, for branching policies, we observe that the query times tend to ramp up after about 4 levels of delegation in the policy, whereas typical user policies such as *"allow friends of friends to see my location"* only use one level of delegation.

Much like the local policy chains, Figure 7(a) demonstrates that the time required to evaluate non-branching remote policy chains is still proportional to the number of links in the chain. Yet, it also shows that there is an added cost due to remote procedure calls that are required when traversing knowledge bases. A similar trend can be seen in Figure 7(b), but in both cases, the time required to evaluate these remote policies does not exceed a few milliseconds. Furthermore these benchmarks are run against policies that are more complex than we would expect a typical user to employ, based on the typical policies that were collected in user studies [24].

The results of the aggregate evaluation shown in Figure 8(a) suggest that aggregate queries scale well even for a large number of audit log records in the database. Finally, the design and evaluation of a real-world policy shown in Table 6.3 support our expectations that query time performance will not be a concern for users of *PlexC*.

7.2 VALIDATION OF USER STUDIES

The design of *PlexC* is motivated by the collection of recent user studies described in Section 2.2. However, these studies share a common limitation: because these studies are based on user surveys, they reveal only the *perceived* preferences and needs of participants in an artificial environment. In other words, although the participants in these studies are likely to have given truthful answers to the surveys, there is a chance that they would behave differently in a real-world scenario. While it is not possible to completely account for all sources of response bias in a lab setting, a field study of a fully functional system would be able to mitigate these effects and support or challenge the findings upon which *PlexC* is based.

7.3 FUTURE WORK

Implementation. We have developed a prototype implementation of *PlexC*, and we are currently integrating it as part of a larger location-sharing application. We have already implemented several major components of the system, including a mobile application to track current location and view the locations of others, a web interface for managing policies and carrying out more complex queries, and a server application to store data and manage social relations. *PlexC* will be used as the fundamental access control component to manage the information flow between the other system components. We plan to continue using this testbed to better understand the utility of *PlexC*, as well as to explore the system design trade-offs present in this space.

Other implementation tasks also deserve further investigation. For example, the study conducted by Biehl et al [5]. revealed that participants were concerned with the storage, format, and and retention policy of their private information. To provide users with ownership of their data, we envision *PlexC* as a component of a distributed network that is capable of running on a user's mobile device.

Usability of Policy Creation. *PlexC* allows users to create concise policies for exposure management, and it inherits many desirable traits from Datalog (e.g., unambiguous semantics and tractable evaluation). However, we do not expect the average user to write rules in *PlexC* directly. *PlexC* was developed, instead, to represent a formal semantics for exposure-aware policies. While it is possible to write *PlexC* policies directly, we envision that most users will interact with their policies via some form of structured policy editor.

At the other end of the spectrum, the user study conducted by Patil et al. [24] demonstrated that participants had difficulty expressing coherent policies in free-form text. While expressing disclosure rules in natural language may certainly be easier for the average user, a large number of policies were ambiguous and unenforceable. We believe that a form-based rule editor would simplify the creation of *PlexC* rules that are easy for users to understand, while still taking advantage of the power of *PlexC*. However, we predict that a user-friendly rule editor might, as a consequence, restrict the flexibility and expressive power of the language. Therefore, striking a good balance between usability and expressive power will be an interesting research challenge.

8.0 SUMMARY

In this thesis we address the development of *PlexC*: a policy language for exposure control. The concept of *exposure* denotes the extent to which an individual's personal data is shared, and addresses the individual's resulting concern for privacy. Given the complexity of this design space, we first articulate requirements for policy languages for exposure control by analyzing the findings of several recent survey studies addressing various facets of the exposure problem. Not surprisingly, existing *access control* policy languages are shown to be insufficient for meeting the *exposure control* needs voiced by participants in these studies. We present *PlexC* as a solution to meet the needs of these participants. After describing the details of *PlexC*, we present a prototype implementation and perform an initial performance evaluation. We show that it is both suitable for meeting the needs of users in modern context-sharing systems, as well as capable of encoding a variety of historically useful policy idioms. Although *PlexC* was derived by examining surveys of users' perceived exposure-control needs, further evaluation work is still required. In particular our team plans to explore the development of usable policy-management interfaces and user studies of *PlexC*-based contextual sharing systems.

Acknowledgments: This research was supported in part by the National Science Foundation under awards CCF–0916015 and CNS–1017229.

9.0 BIBLIOGRAPHY

[1] Michael Backes, Matteo Maffei, and Kim Pecina. A security api for distributed social networks. In *NDSS*, February 2011.

[2] Moritz Y. Becker, Cedric Fournet, and Andrew D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 2009.

[3] Moritz Y. Becker and Sebastian Nanz. A Logic for State-Modifying Authorization Policies. *ACM TISSEC*, 13:20:1–20:28, July 2010.

[4] Moritz Y. Becker and Peter Sewell. Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *POLICY*, pages 159–168, June 2004.

[5] Jacob T. Biehl, Eleanor Rieffel, and Adam J. Lee. When Privacy and Utility are in Harmony: Towards Better Design of Presence Technologies. *Personal Ubiquitous Computing*, in press, February 2012.

[6] Nick Bilton. Burglars Said to Have Picked Houses Based on Facebook Updates. http://bits.blogs.nytimes.com/2010/09/12/burglars-picked-houses-based-on-facebook-updates/, 2010.

[7] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Infrastructures (Position Paper). LNCS 1550*, pages 59–63, 1998.

[8] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *In*

Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 164–173, May 1996.

[9] Anthony J. Bonner. Transaction Datalog: a Compositional Language for Transaction Programming. In *In Proceedings of the International Workshop on Database Programming Languages*, 1997.

[10] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE TKDE*, 1:146–166, March 1989.

[11] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY*, pages 18–38, 2001.

[12] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in dlv. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 847–852, 2003.

[13] John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.

[14] Carl A. Gunter, David M. Liebovitz, and Bradley Malin. Experience-based access management: A life-cycle framework for identity and access management systems. *IEEE Security & Privacy Magazine*, 9(5), September/October 2011.

[15] Adam J. Lee, Ting Yu, and Yann Le Gall. Effective trust management through a hybrid logical and relational approach. In *ASIACCS*, April 2010.

[16] Jiangtao Li, Ninghui Li, and William H. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 46–57, New York, NY, USA, 2005. ACM.

[17] Ninghui Li and John C. Mitchell. RT: A role-based trust-management framework. In

Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX III), pages 201–212, April 2003.

[18] Locaccino. http://locaccino.org/.

[19] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application. In *SenSys*, pages 337–350, 2008.

[20] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.

[21] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *CCS*, pages 289–298, 2008.

[22] Jaehong Park and Ravi Sandhu. Towards usage control models: beyond traditional access control. In *SACMAT*, pages 57–64, 2002.

[23] Jaehong Park and Ravi Sandhu. The uconabc usage control model. *ACM TISSEC*, 7(1):128–174, February 2004.

[24] Sameer Patil, Yann Le Gall, Adam J. Lee, and Apu Kapadia. My Privacy Policy: Exploring End-user Specification of Freeform Location Access Rules. In *Proceedings of the Workshop on Usable Security (USEC)*, March 2012.

[25] Sameer Patil, Greg Norcie, Apu Kapadia, and Adam J. Lee. Reasons, Rewards, Regrets: Privacy Considerations in Location Sharing as an Interactive Practice. In *SOUPS*, July 2012.

[26] Norman Sadeh, Jason Hong, Lorrie Cranor, Ian Fette, Patrick Kelley, Madhu Prabaker, and Jinghai Rao. Understanding and Capturing People's Privacy Policies in a Mobile Social Networking Application. *Personal and Ubiquitous Computing*, 13:401–412, August 2009.

[27] Ravi Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[28] Roman Schlegel, Apu Kapadia, and Adam J. Lee. Eyeing your Exposure: Quantifying and Controlling Information Sharing for Improved Privacy. In *SOUPS*, July 2011.

[29] Janice Y. Tsai, Patrick Kelley, Paul Drielsma, Lorrie Faith Cranor, Jason Hong, and Norman Sadeh. Who's viewed you?: the impact of feedback in a mobile location-sharing application. In *ACM CHI*, pages 2003–2012, 2009.

[30] Walt Yao, Ken Moody, and Jean Bacon. A Model of OASIS Role-Based Access Control and its Support for Active Security. In *SACMAT*, pages 171–181, 2001.